arXiv:2301.02308v1 [cs.PL] 5 Jan 2023

美国　高级类型系统的可用性：以 Rust 为例

KASRA FERDOWSI,加利福尼亚大学圣迭戈分校，美国

能够强制保证各种正确性和安全性的高级类型系统——例如线性类型和所有权类型——在编程语言研究领域已有悠久的历史。尽管如此，对于这些类型系统及其可用性的以人为中心的评估几乎一直付之阙如；实证评估仅限于测试专家（即类型系统的创建者）编写的程序所具备的表达能力。

过去几年，随着仿射类型和所有权的一种变体在广受欢迎的 Rust 编程语言中得到采用，这一情况开始改变。随着 Rust 日益流行，各种研究开始对 Rust 的所有权与生命周期规则的可用性进行实证评估，并提供了关于此类类型系统可用性的丰富定性和定量信息。研究发现，尽管 Rust 总体上成功实现了其安全性和性能方面的承诺，但这些规则的学习曲线十分陡峭，并且一再被认为是阻碍人们采用 Rust 的因素。

在本报告中，我简要介绍线性类型和基于区域的内存管理的历史，这些概念直接启发了 Rust 的类型系统。随后，我将介绍 Rust 的所有权和生命周期规则，并综述学术研究在评估这些规则的可用性方面的最新进展。我将讨论这些规则为何难以学习和应用的理论论证与实证证据，并概述针对其中部分难点的现有研究。我还将借鉴人机交互和计算机科学教育领域的更广泛研究，提出该领域未来的研究方向。

1 引言

尽管编程语言研究领域对高级类型系统进行了大量研究，从依赖类型 [Xi 和 Pfenning 1999] 到线性类型 [Wadler 1990] 以及所有权类型 [Clarke 等人 2013], 但这类类型系统很少跨越学术界的边界，进入主流通用编程语言。

这或许是其带来的副作用，也可能因编程语言领域过去对以人为中心的方法缺乏兴趣而加剧 [Coblenz 等人 2018], 其结果是，此类类型系统几乎没有值得注意的用户评估。因此，尽管人们反复讨论其可用性，关注点却在于某个特定类型系统是否“富有表现力”，也就是说，专家能否编写出复杂且实用、并能通过该系统类型检查的代码。这意味着，直到最近，我们才终于开始了解这类类型系统是否容易学习，以及非专家会如何学习和使用它们。

这种情况随着 Rust 编程语言的出现开始发生变化 [Klabnik and Nichols 2017; Matsakis 和 Klock 2014]。Rust 将“所有权”“借用”和“生命周期”的概念作为一种类型系统来实现，使其能够在编译时保证内存安全和线程安全，而无需垃圾回收。随着 Rust 日益成为一种广受欢迎的通用编程语言，针对其所有权模型可用性的新一轮研究也随之兴起。

在本报告中，我旨在利用这项新研究，更好地理解高级类型系统的可用性，并探讨主流软件工程师是否以及如何采用这些系统。本报告的其余部分组织如下：第2节简要介绍与 Rust 最相关的类型系统的发展历史。第 3 节随后介绍 Rust 对这些类型系统的具体实现，第4节综述评估和改进 Rust 中所有权可用性的相关工作。最后，第5节结合这些研究的要点，以及人机交互和计算机科学教育领域的相关理论，探讨高级类型系统可用性研究未来可能的方向。

研究考试，*2022*年*4*月*27*日，美国加利福尼亚州拉霍亚2022年。

2 背景

本节简要概述线性类型、基于区域的内存管理（RBMM）和所有权类型，随后从更高层次讨论它们之间的共同主题。这一讨论并不旨在面面俱到，因为每个系统都有着悠久的历史，单独对任何一个系统进行完整介绍都需要一篇专门的综述论文[1]。相反，本节旨在为启发 Rust 的相关理论提供背景，并帮助更好地将 Rust 的可用性研究发现与更广泛的类型系统联系起来。

2.1 线性类型

受 Girard 的线性逻辑启发 [, Jervell 1996],线性类型由 [Wadler 1990] 提出，旨在让函数式编程语言能够安全地“改变世界”（即修改状态）。线性类型的核心是必须“恰好使用一次”的值，也就是说，这些值不能被复制或隐式丢弃。Wadler 指出，这项限制能够实现多种静态检查和功能，对内存管理和程序正确性非常有用。

更具体地说，他指出，线性类型无需垃圾回收即可管理可变值的内存。如果一个值不能被复制或隐式丢弃，并且必须恰好使用一次，那么在使用完之后，我们就可以回收它占用的内存。这既能处理内存管理问题，也能防止释放后使用和重复释放错误。通过禁止别名，线性类型还解决了推理修改行为的问题，无论是在单线程代码中（别名引用是一个显著的错误来源），还是在多线程代码中（别名可能导致竞态条件）。

正如这些类型系统中一个常见的主题，Wadler 也指出，严格的线性类型约束过强，并非必需。他引入的语言并非严格的线性语言，而是允许对线性值进行多次“读访问”（不可变引用）；一旦存在“写”访问（可变引用），该值便不能再被使用。我将在第 3 节进一步讨论线性类型与 Rust 的关系，但 Rust 采用的也是一种比线性性更宽松的概念，称为仿射类型。具有仿射类型的值并非必须恰好使用一次，而是至多使用一次，也就是说，它可以被忽略 [Pierce 2004]。

虽然这里不会深入讲解，但值得一提的是，线性类型已在多项研究中得到实现和扩展，从其在命令式编程语言 Vault 中的实现 [Fahndrich 和 DeLine 2002] 到近期被加入 Haskell[Bernardy 等人 2017]。它们也启发了本节介绍的其他类型系统。基于区域的内存管理 [Tofte 和 Talpin 1997] 以及所有权类型 [Clarke 等人 1998], 的开创性研究都提到了线性性，而且线性性也随着时间推移影响了这些研究：[Walker 和 Watkins 2001] 将其与区域类型结合，而所有权转移这一概念则结合了线性类型和非线性类型 [Clarke 等人 2013]。

2.2 基于区域的内存管理

顾名思义，基于区域的内存管理（RBMM）是一种利用类型系统进行静态内存管理的尝试。RBMM 最初是效应类型系统的扩展 [Pierce2004] 于 [Tofte 和 Talpin 1997]。不过，它在 Cyclone 语言中的实现 [Grossman 等，2002a,b； Jim 等 2002] 对 Rust 的影响更为直接，因此我将在此重点讨论这一点。

Cyclone 最初是类型化汇编语言项目的一部分 [Morrisett 等人 1999], 但后来发展成为一个独立项目，旨在成为“C语言安全方言” [Grossman 等人2005]。因此，除了基于区域的内存管理之外，它还包含许多有趣的设计选择和语言特性，例如带标签联合体、空值检查和存在类型。 [Grossman 等人 2005] 提供了一个

[1]这实际上存在于所有权类型的情形中 [Clarke 等 2013]

对所有这些特性的简明介绍，但本文将重点讨论 [Grossman 等，2002a], 中所描述的基于区域的内存管理，因为它与本报告最为相关。

RBMM 的核心思想是，将程序中词法作用域限定的一部分与一个命名的“区域”（一种可动态增长的内存部分）关联起来，并使用该区域的名称标注指向该区域的指针类型。这样，编译器就能在作用域结束时自动释放整个区域，而类型检查器则能保证，指向某个区域的指针不会在该区域的作用域之外（即该区域被释放后）进行解引用。为此，类型系统会跟踪程序中每个位置处仍处于活动状态的区域集合（称为该位置的“能力集”），并禁止进行指针解引用，除非该指针对应的区域位于能力集中。

这是 RBMM 的核心，但要使其具备足够的表达能力并保证可靠性，还涉及许多更微妙的细节。例如，区域类型可以支持 子类型。由于区域可以嵌套，指向外部区域的所有指针都保证在内部区域存续期间保持有效（因为外部区域会在内部区域之后释放）。因此，如果某个区域 `'a` 包含一个更小的区域 `'b`，那么标注为 `'a` 的指针是 `'b`的子类型。

Cyclone 的开发者还考虑了区域注解带来的语法负担，以及对区域泛型（参数和返回类型可针对区域注解进行泛型化的函数）的需求。他们的解决方案结合了过程内区域注解推断（免去了在函数体中显式写出大多数注解的需要）以及对部分注解函数签名的默认规则。这大幅减轻了语法负担，并使某些函数可以直接从 C 移植到 Cyclone，无需修改或只需少量修改。

除 Cyclone 外，基于区域的内存管理由来已久。它已应用于 Go 编程语言 [Davis 等人 2012], 实时 Java [Boyapati 等人，2003b], Prolog [Makholm 2000],GPU 编程 [Holk 等人 2014], 以及大数据系统 [Nguyen 等人 2015]。但这些研究都没有通过实证评估其系统对程序员的可用性，而是侧重于基准测试性能，并将对可用性的讨论局限于表达能力和语法负担。

2.3 所有权类型

尽管名称相似，所有权类型与 Rust 中的所有权概念并无直接关系。不过，它们与 Rust 有许多共同目标，也是所有权这一概念发展史上的重要组成部分。因此，若不讨论所有权类型，关于 Rust 相关类型系统的介绍便不完整。

所有权类型是面向对象编程（OOP）发展而来的，旨在通过静态方式强制执行更严格的“封装”概念 [Clarke 等人， 2013]。尽管不同实现的具体细节差异很大，但其基本思想是在类型系统中编码对象之间的“拥有”与“所有者”关系。这会对指针别名施加限制，从而强制实现封装 [Clarke 等人， 1998], 并支持额外的检查与保证，例如防止数据竞争和死锁 [Boyapati 等人， 2002] 以及悬空指针 [Boyapati 等人，2003b]。

尽管关于所有权类型的研究成果十分丰富，而且它与 Java（一种流行的通用编程语言）关系密切，但这些类型既未得到广泛采用，也未在真实用户中进行评估。相反，这些类型的每项扩展、实现或应用都仅由系统设计者进行评估；他们使用自己的类型系统编写现实世界中的应用，以论证其表达能力 [Aldrich 等人，2002；Boyapati 等人，2003a，b；Clarke 等人，2013，1998]。

尽管其发展史大多独立于其他类型系统，但所有权类型在概念上与本报告中的其他类型系统十分接近。例如，在所有权类型中，“程序的堆被划分为分层嵌套区域，最初称为所有权上下文” [Clarke 等人 2013] 这

与基于区域的内存管理中的区域类似。事实上， [Boyapati 等人， 2003b] 将所有权类型与基于区域的内存管理相结合，为 Java 实时规范（RTSJ）实现了一种类型系统。该系统在静态层面保证了悬空指针运行时检查能够通过，并确保不存在对垃圾回收堆的引用（这是 RTSJ 代码的一项要求）。然而，与本节中的其他研究类似，该系统从未在真实用户中进行过评估。

2.4 共同主题

到目前为止，我分别介绍了每种类型系统，但这些理念和系统彼此密切相关，其发展也难以截然分开。因此，在转向 Rust 之前，我将先更全面地讨论这些研究中的总体趋势。

线性类型、区域类型和所有权类型都关注内存管理与安全性。它们都认识到，类型系统可以帮助程序员推理复杂程序，避免因使用别名引用或已释放引用而产生的各种错误，并在无需运行时检查或垃圾回收的情况下，为内存管理提供可证明正确的解决方案。

它们也都很快注意到并尝试解决类型系统的“表达能力”与其保证之间的权衡。在介绍线性类型的论文中， [Wadler 1990]并未强制要求线性性，而是允许组合线性类型和非线性类型的值；并且（如上所述）放宽了唯一性的定义，从而允许对线性值存在多个只读引用。尽管推崇基于区域的内存管理，Cyclone [Grossman 等人， 2002b] 仍包含一个专门的垃圾回收堆区域。所有权类型问世几年后，其早期倡导者便已开始论证，严格的唯一性限制过于苛刻 [Clarke 和 Wrigstad 2003]。

最后，尽管这些研究关注表达能力，却倾向于将其类型系统实现为流行编程语言的变体或扩展（ML [Tofte 和 Talpin 1997], C[Grossman 等人， 2002b], Java [Aldrich 等人 2002], Scala [Haller 和 Odersky 2010], Go [Davis 等人 2012], Prolog [Makholm 2000], 等），甚至考虑到这类限制对程序理解的益处 [Aldrich 等人 2002；Clarke 等人 2013], 上述所有研究都缺少以用户为中心的方法。没有人研究过，除类型系统的发明者和实现者之外的用户，能否轻松地应对这些类型系统施加的限制。

3 Rust 编程语言

Rust 虽然受到 [Wadler 1990] 关于所有权概念的启发，并且 [受到 Grossman 等人， 2002b]关于生命周期和内存管理方法的启发，但并未直接实现上述任何一种类型系统。相反，它将这些系统与许多其他理念结合起来，力求在不使用垃圾回收的情况下确保内存安全和线程安全，同时实现静态内存管理。然而，它也旨在成为一种通用系统编程语言[2], ，因此追求“实用主义安全” [Evans 等人 2020] ，并提供了一些功能，可绕过静态检查，以获得更好的性能或支持更复杂的别名模式。

在本节接下来的内容中，我将首先介绍 Rust 中的所有权，然后说明生命周期及其与所有权的结合方式，最后介绍绕过其中一些检查的不安全代码。这里的介绍仅限于理解我将在第4节讨论的一些可用性问题所必需的内容；我会略去类型系统的许多细微之处，也不会提及 Rust 的语法、语义等。若想全面了解 Rust，我推荐阅读

[2]“系统编程语言”这一术语最近引发了一些争议 [Crichton 2018]。不过，鉴于该术语通常指编译为汇编语言并提供底层资源控制能力的语言（与 Python 这类解释型语言，或 Java 这类运行在更高抽象层级的语言相对），为求简便，我将在本报告中使用这一术语。

```
1  让v = vec![1 , 2];
2  令v2 = v;
3
3  打印(&v);
```

(a)

```
1  let v = vec![1, 2];
2  设x = &v[0];
3  令v2 = v;
4
4  令y = *x + 1;
```

(b)

```
1  let mut v = vec![1, 2];
2  让x = &v[0];
3  Vec::push(&mut v, 3);
4  设y = *x + 1;
```

(c)

图 1。每个子图都展示了对第 3.1 节中相应规则的违反。在图 1a 中，向量的所有权在第 2 行转移给了v2，因此v 无法在第 3 行调用print 时被借用。在图 1b 中，x 是对v 的引用，并持续到第 4 行最后一次使用。但v 只持续到第 3 行将其向量转移给v2 为止。同样，在图 1c中，对v 的不可变借用位于x 中，从第 2 行持续到第 4 行最后一次使用，因此v 无法被第 3 行对*push* 的调用以可变方式借用。

Rust 官方书籍 [Klabnik 和 Nichols 2017], , 这本书内容全面，且非常易读。

3.1 所有权

Rust 中的所有权规则在纸面上相当简单，也有多篇论文试图对其进行总结 [Fulton 等人，2021；Qin 等人 2020]。本文将采用 [Crichton 2020], 的总结，因为它最为简单、简洁。但首先，我需要介绍一些术语。

在 Rust 中，每个值（字符串、i32、Vec 等）都由一个变量所拥有，该变量就是其所有者。由于直接操作值可能不太方便，Rust 还提供了值的引用，可以从所有者那里借用该值。最后，在 Rust 中，变量和引用默认都是不可变的。若要通过变量或引用修改值，就需要使用 mut 关键字将其显式标记为可变。例如，在图 1c 第 1 行中，调用 vec! 创建的 Vec 被赋给变量 v，因此 v 拥有该 Vec。在第 2 行中，x 是对 v 的第一个元素的引用，因此 x 借用了 v。最后，v 使用 mut 关键字标记，因此它是可变的；但 x 没有，因此 x 只能被读取，不能被修改或重新赋值。

考虑到所有权、引用和可变性， [Crichton 2020] 这样总结 Rust 的所有者规则：

> (a) 每个值都恰好有一个所有者。(b) 对值的引用不能比其所有者存活得更久。(c) 一个值可以有一个可变引用，或多个不可变引用。

有关违反这些规则时可能出现的代码示例，请参见图 1 。Rust 的编译器rustc 包含一个常被俗称为借用检查器的检查阶段；如果它无法静态确定所有这些规则都得到遵守，就会报错。

在继续讨论生命周期之前，值得考虑一下这些规则与第 2 节中的类型系统有何关联。颇为反直觉的是，Rust 的所有权规则与第 2 节[3]中讨论的所有权类型并无直接关系。相反，第一条规则通常被称为“所有权”，与线性类型的联系最为密切：程序中的任意时刻，一个值都只有一个所有者；虽然所有权可以在变量之间转移，但不能隐式复制。

这并不是说所有权类型完全无关。与线性类型或区域类型相比，所有权类型对可变性和借用的处理更加明确。某些所有权类型的实例将值的变更限制为只能由所有者进行，并且只允许其他对象进行读取访问。类似地，所有权类型文献的部分内容中也出现了具有类似功能的借用概念 [Aldrich 等人 2002；Boyland 2001]。但所有权类型与之紧密相关

[3]近期的一些论文进一步加剧了这种混淆，例如 [Crichton 等人 2021] 的论文将 Rust 的所有权规则称为“所有权类型”，尽管“所有权类型”作为一种不同的系统早已有其发展历史。

```
1 结构体 Foo<'small,'large:'small> { 10 fn main() { 2 a: &'small str, 11 let mystr = "abc"; 3 b: &
'large str, 12 let substr = &mystr[0..2]; 4 } 13 let foo = Foo::new(&mystr, substr); 5 impl<'s, 'l: 's>
Foo<'s, 'l> { 14 } 6 fn new(x: &'s str, y: &'l str) ->Self { 7 Self { a: x, b: y } 8 } 9 }
```

图 2. `struct` 和 `fn` 定义中显式生命周期参数的示例。生命周期参数将 `struct` 和 `fn` 定义为生命周期 `'small` 和 `'large`的泛型，其中 `'large` 是 `'small`的子类型。随后 Rust 会使用这些参数，通过比较传递给 `Foo::new` 的引用推断出的生命周期与显式生命周期参数，对 `main` 函数中的 `Foo::new` 的使用进行类型检查。这段代码能够编译，因为 `substr` 借用了 `mystr`，因此其生命周期必须短于 `mystr`的生命周期，这满足了 `'small`和 `'large`之间的子类型要求。

在面向对象编程（这并非 Rust 所采用的范式）中的概念。而在相关文献中，我只发现一处提及所有权类型对 Rust 的影响 [Weiss 等人2021]。

我会在下一节详细比较基于区域的内存管理，但请注意，规则 (a) 和 (b) 也允许自动内存管理：当一个值的所有者离开作用域时，可以保证该值不再有任何存活的引用，因此编译器可以插入对该值调用释放内存的代码（`drop`，用 Rust 的术语来说）。这能在无需垃圾回收的情况下保持内存安全。

3.2 生命周期

[Fulton 等人 2021] 对生命周期作出了简洁而出色的描述：

> “生命周期命名一个作用域，而引用上的生命周期标注则告诉编译器，该引用仅在该作用域内有效。”

这里所说的生命周期标注，是指 Rust 引用不同于 C 风格的“原始”指针。原始指针的类型只包含其所指向值的类型。而 Rust 引用的类型则会标注一个生命周期，指明该引用在哪个作用域内有效，并让借用检查器跟踪它正在借用哪个值（或其他引用）。

Rust 会自动推断函数体中的所有生命周期，因此大多数生命周期标注对用户不可见。不过，在编写签名中包含引用的函数，或存储引用的数据类型时，Rust 要求用户显式地将其写成针对这些引用生命周期的泛型函数或数据类型[4]。示例见图 2。

关于生命周期、它们的计算方式，以及它们对表达能力和可用性的影响，还有许多内容值得探讨。但正如我们将看到的，Rust 的生命周期以复杂难懂而著称，对这些方面的完整描述超出了本报告的范围。因此，生命周期就先讨论到这里，本节最后将探讨它们与 Cyclone 的关系。

读者现在可能已经注意到，Rust 中的生命周期与 Cyclone 中的区域和区域注解非常相似，包括它们的语法、子类型和过程内推断。事实上，

[4]但有一种例外：某些函数的签名符合特定模式，例如不返回引用的函数，或恰好接收一个引用并返回一个引用的函数。在这些情况下，Rust 会省略这些生命周期，以简化语法。请注意，这与生命周期推断并不相同。与 Cyclone 类似，Rust不会在函数签名中推断生命周期。

Rust 的许多特性都受到 Cyclone 的启发，例如自动边界检查和和类型（Rust **枚举**以及 Cyclone 的带标签联合体）。

两者的主要区别在于，Cyclone 要求手动定义区域的语法作用域，并使用区域名称（一等值）手动分配值并在不同区域内初始化值。在 Rust 中，生命周期由编译器自动确定，不能显式设置或使用（泛型生命周期参数除外）。此外，从 Rust 2018 开始，Rust 的区域不再由词法作用域确定，而是根据程序的中间控制流图表示进行计算 [Rust 核心团队 2018]。因此，虽然区域和生命周期背后的理论相同，但它们的实现和交互模型在许多有趣且重要的方面有所不同。

3.3 不安全 Rust

第 2 节 所讨论的类型系统有一个共同特点：它们都发现，严格遵守各自的规则会造成不必要的限制，并设法放宽规则以提升表达能力。Rust 也不例外，不过它的解决方案相当不同。

Rust 所有权规则的一个重要问题是：它们是健全的，但不可判定。因此，借用检查器并不完备；有大量遵循所有权规则的安全代码，借用检查器却无法通过静态分析加以验证。为缓解这一问题，Rust 允许用户显式将函数和代码块标记为**不安全**，并在这些不安全代码块中禁用某些安全检查[5]。具体来说，**不安全**允许代码执行以下操作：

- 解引用裸指针
- 调用不安全函数（包括 C 函数、编译器内建函数和裸分配器）
- 实现不安全特征
- 修改静态变量
- 访问联合体字段

[Rust 项目开发者 2022]。这仍然相当严格，但会带来重要影响。例如，通过将具有某个生命周期的引用转换为裸指针，再解引用该裸指针，将其重新借用为一个具有新生命周期的引用，就可以绕过所有权规则。这允许同时为一个值创建多个可变引用，从而违反所有权的第三条规则。

这方面的细节一如既往地复杂且超出了本报告的讨论范围，但不安全代码对于 Rust 的“实用主义安全”至关重要。它既能实现一些借用检查器无法从底层进行推理的性能优化，也能与外部代码交互，还能支持某些程序员能够验证为安全、但无法通过借用检查器检查的别名模式。

4 所有权的可用性

截至本文撰写时，已有五篇关于 Rust 所有权类型系统可用性的主要论文 [Coblenz 等人，2021；Crichton 2020；Fulton 等人，2021；Zeng 和 Crichton，2019；Zhu 等人2022]。此外，遵循行为研究的最佳实践 [Mcgrath 1995]，，这些研究采用多种方法，探讨若干相似且相互交叠的研究问题。因此，本节不会逐篇评述，而是综合讨论这些研究的发现，并在相关内容中介绍各篇论文及其研究方法。我还会借鉴有关 Rust 中**不安全**用法的研究，以及关于程序员如何学习新编程语言的相关研究。

[5]一些人，[例如 Zhu 等人，2022] 将不安全代码描述为“类似于 C 编程语言”。从技术上说，这确实如此，因为不安全代码可以与任意 C 代码交互；不过，Rust 中的不安全代码仍比 C 严格得多。

总体而言，Rust 的所有权模型确实难以学习，其设计的某些方面即使对经验更丰富的 Rust 开发者来说仍然很难理解。然而，Rust 所承诺的安全性和性能，加上出色的工具链以及与其他语言互操作的功能，使 Rust 依然广受欢迎，也深受成功采用它的人喜爱。

4.1 所有权难以使用吗?

> “学习 Rust 所有权就像在一座墙壁由石棉和挫败感构成的迷宫中穿行，而这座迷宫没有出口；每次你走到死胡同，都会患上动脉瘤并死去。”
>
> — 来自 [Coblenz 等人 2021]

Rust 因其“陡峭的学习曲线”而臭名昭著，这一点也被认为是其在工业界采用过程中面临的主要问题。然而，有趣的是，研究表明，即使是经验丰富的开发者，在 Rust 所有权的某些方面也会遇到困难。

4.1.1 Rust 新手程序员面临的障碍。 [Zeng 和 Crichton 2019] 对 Reddit 子版块 `/r/rust` （一个专注于 Rust 的在线社区）的热门帖子，以及 Hacker News（一个范围更广的科技论坛）上的文章及其对应评论进行了内容分析，以确定采用 Rust 的障碍。在他们审阅的 18 篇经验报告和语言比较中，他们发现“借用检查器的复杂性是被提及频率第二高的抱怨”（仅次于编译器版本问题）；借用检查器不允许其他语言中常见的内存访问模式，因而令人沮丧。

[Fulton 等人 2021] 随后对 16 位曾尝试在其生产团队中采用 Rust 的行业专业人士进行了访谈，并根据访谈结果设计了一项在线调查，另外获得了 178 位参与者。他们也发现，Rust 陡峭的学习曲线是采用 Rust 的最大障碍。

请注意，这种困难不仅仅是学习一门新语言的困难，也不仅仅是在没有垃圾回收的情况下学习一门系统编程语言。 [Fulton 等人 2021] 发现，学习 Rust 最大的挑战具体在于借用检查器，以及为编写能够通过借用检查器检查的代码而必须进行的编程范式转变。并且 [Shrestha 等人 2020] 引用了一位 C++ 开发者的话，他表示借用检查器“迫使程序员以不同的方式思考”。因此，Rust 更先进的类型系统似乎才是其难点的主要来源，而不仅仅是缺乏垃圾回收或需要进行更多底层编程。

这并不是说加入垃圾回收就不能降低 Rust 的使用难度。 [Coblenz 等人 2021] 对一门大二级别编程课程中的 428 名学生开展了对照研究。学生们先接受了为期两周的 Rust 课程讲授，之后被要求完成一项作业，该作业需要他们充分理解 Rust、其所有权规则，以及支持内部可变性的类型。研究人员将学生随机分成两组：一组使用 Rust 标准库的数据类型完成作业，另一组则使用一种垃圾回收包装类型（名为“Bronze”）。这种包装类型允许通过借用检查器的别名模式更多，从而免去了使用更复杂的别名模式和数据类型的需要。

他们发现，在完成率和完成作业所需的自我报告时间方面存在显著差异。平均而言，使用 Bronze 的学生所花时间仅为对照组的三分之一，完成作业的可能性约为对照组的 2.44 倍。有趣的是，组间的时间差异只出现在任务的第二部分，该部分涉及复杂的别名和可变性。作业的第一部分只侧重于所有权，组间并未表现出显著差异。

4.1.2 经验丰富的 *Rust* 程序员面临的障碍。除了最初的学习曲线，研究还表明，即便是经验丰富的开发者，所有权的某些方面仍然难以运用。在关于 Rust 中内存安全和线程安全问题的研究中，[Qin 等人 2020] 检查了五个 Rust 系统和应用程序、五个热门库以及两个漏洞数据库。他们发现，这些代码库中导致漏洞难以修复的一个常见原因，是“对 Rust 生命周期规则缺乏良好的理解”。这一点值得注意，因为与上述研究的参与者不同，在这些代码库中工作的程序员理应是经验丰富的 Rust 开发者。

Rust社区2020年调查的结果也印证了这一发现 [Rust 调查团队 2020]。他们收到了8323份答卷，参与者中自评专业水平为7分（满分10分）的人数最多。他们还发现，生命周期是最难学习的主题，但遗憾的是，他们没有报告这一回答是否以及如何随专业水平评分而变化。类似地，[Zhu 等人 2022] 研究了 StackOverflow 上关于 Rust 所有权规则的100个问题样本，发现这些问题中最常见的安全规则违规原因是“复杂的生命周期计算”，共出现了74次[6], 44，其中过程内生命周期计算出现16次，显式生命周期参数出现16次，省略的生命周期参数出现14次。

从这些研究来看，可以有把握地得出结论：Rust 的所有权规则确实难以学习。它们严重阻碍了人们学习和采用 Rust；即使对于经验更丰富的 Rust 开发者而言，理解生命周期仍然是一个难题。

4.2 为什么所有权难以使用？

> “我可以在一堂课内向一屋子的本科生讲清楚所有权的三条规则 [] 。但每次使用 Rust 时，借用检查器变幻莫测的行为还是会让我栽跟头！”
>
> — [克赖顿 2020]

如果我们承认 Rust 确实比同类系统编程语言更难学习，而且这在很大程度上是由其所有权类型系统造成的，那么接下来就要探究：Rust 的所有权规则究竟有哪些难以学习和应用之处。

4.2.1 范式转变。一种解释或许是 Shrestha 等人 [所使用的“干扰”这一概念 2020]。在那篇论文中，他们对 StackOverflow 上涉及 18 种不同编程语言的 450 篇帖子进行了定性编码，并采访了 16 位专业程序员，以了解经验丰富的开发者如何学习新的编程语言，以及他们在学习过程中遇到哪些困难。他们借用了心理学和神经科学中的“干扰”一词来阐述这项研究[7]。这个词指“先前的知识妨碍对新学信息的回忆”。这可能只是两种语言在零起始索引与一起始索引上的差异，也可能涉及更大的不同：用新语言编程需要“思维转变”，也就是从根本上改变范式。

学习 Rust 需要这样的思维转变，因为它的所有权规则禁止许多常见的编程模式。以双向链表为例。在大多数语言中，实现起来几乎不费吹灰之力，但从定义上说，它违反了 Rust 的规则：它至少需要对一个节点有两个可变引用，一个来自前一个节点，一个来自后一个节点。Rust 提供了应对此问题的方法，最简单的就是使用具有“内部可变性”的数据类型，将同时可变访问的检查推迟到运行时，但这些方法更难学习和使用。因此，[Shrestha 等人 2020] 以 Rust 的所有权类型系统为例说明这种思维转变，并引用了一位 C# 开发者的话：他必须“彻底重新思考自己通常会在 C# 中解决的问题”。

[6]论文统计了 77 个违规，但我排除了其中 3 个，因为它们仅仅是语法错误。[7]“促进”一词也被排除在外，不过它与这里的内容不太相关。

```
let mut v = vec![1, 2];
令 一 = &可变v[0];
令二 = &可变v[1];
*两个 += *一个;
```

(a)

```
let mut v = vec![1, 2];
让迭代器 = v.迭代器_可变();
设一个 = iter.next().unwrap();
让两个 = iter.next().unwrap();
*两个 += *一个;
```

(b)

```
let mut v = vec![1, 2];
v.insert(0, v[0]);
v.get_mut(v[0]);
```

(c)

图 3。图 图 3a 和 图 3b 中的程序实现了相同的功能，但只有图 3b 通过了借用检查器。在 图3c中，第 2 行和第 3 行的语句在类型层面几乎完全相同，但只有第 2 行通过了借用检查器，这大概是由某些实现细节导致的。

在研究的定性部分中，[Coblenz 等人 2021] 在学生的调查回复中也发现了类似的主题。对于没有使用 Bronze 库的学生，作业的第二部分要求使用具有内部可变性的数据类型，并显式声明生命周期参数。学生提到了使用这些类型的困难，以及为了正确使用它们而需要重新设计代码的问题，这使论文作者得出结论：“垃圾回收的大部分益处来自架构简化”，并且“设计是非 Bronze 参与者与 Bronze 参与者之间性能差异的重要成因”。

因此，至少有一个导致所有权难以学习的主要原因是它要求进行范式转变。初学 Rust 的程序员需要学习全新的模式，并在架构层面以新的方式组织代码。而他们以往的经验可能会主动妨碍学习，因为他们需要摒弃常见的编程模式，学会以新的、违反直觉的方式组织代码。

4.2.2 错误消息。 [Coblenz 等人 2021] 还指出，`rustc` 的错误消息加剧了困惑和挫败感。`rustc` 的错误消息不仅描述代码中的错误，对于某些错误模式，还会建议可能修复问题的修改。不过，这些修改始终局限于局部，无法提供任何有助于实现思维转变的高层次设计反馈。最好的情况下，这些建议会让学生进行一连串局部修改，最终得到能够编译的代码，却不明白其中的原因。最糟糕的是，正如一名学生发现的那样，这些建议可能会循环往复，“比如删除 &，然后删除后又尝试添加 &。”这使作者得出结论：Rust 的错误消息并不能“帮助设计或理解”。

[Zhu 等人 2022] 更直接地研究了错误消息。他们运用认知任务分析 [Diaper 2004] ，了解专家如何解决他们在一组 StackOverflow 问题中发现的 110 个 Rust 所有权错误，并将专家采取的步骤与错误消息中包含的信息进行了比较。他们发现，对于大多数错误，错误消息都包含了所有相关信息；但对于 32 个错误，消息未能解释“计算生命周期或借用关系的关键步骤”；另有 10 个错误未能 “ 解释两个生命周期注解之间的关系 ”，还有 9 个未能说明 “ 安全规则如何作用于特定代码结构”。

关于`rustc` 错误消息的更广泛讨论，我会留到第 5 节再谈；不过，此处引用的研究表明，Rust 错误消息并未提供必要的帮助。程序员的错误可能具有更深层的结构性，但错误消息只会建议可能造成误导的局部修改。即使是局部错误，错误消息也不总是包含理解并修复错误所必需的信息，而且还假定程序员具备一些外部知识。但这仍然无法解释，为什么一位有经验的Rust 开发者会在所有权问题上遇到困难。

4.2.3 不完备性之咒。 [克莱顿 2020] 指出，所有权规则简单易学，但要静态地检查这些规则，就像检查“大多数有趣的程序属性”一样，是不可判定的。因此，Rust 在借用检查器中实现这些规则时，必然

是不完备的，而所有权方面的许多可用性问题都源于程序员对规则的理解与借用检查器验证这些规则的能力之间的差距。

请看图3a 和图3b中的例子。这两个程序执行的功能相似：获取向量中两个元素的引用，并将其中一个元素的值加到另一个上。但只有第二个程序能够编译，因为借用检查器无法推理索引。它保守地认为，图3a 中的第 2 行和第 3 行都在可变借用整个向量，因此违反了第 3.1 节中的规则（c）。然而，[8], 使用了迭代器，借用检查器可以在类型层面进行推理，并成功验证其没有违反所有权规则。因此，图3b 能够成功编译。请注意，这两个程序都是“安全的”；理论上，涉及依赖类型的更高级类型系统可以静态验证图3a的安全性，但当前类型系统的局限意味着，开发者不仅需要学习所有权规则，还需要了解借用检查器如何验证这些规则。

这个问题因借用检查器的实现相当复杂而更加突出，有时看似非常相似的代码也可能因为晦涩的原因而无法编译。代码示例 图3c中有两个相似的向量函数调用。第 2 行获取 `v` 的第一个元素，并将其插入 `v`，使其成为新的第一个元素。第 3 行则使用第一个元素的值 作为索引，以获取 `v` 中第二个元素的可变引用。这些函数的类型非常相似：两者都通过对 `v` 进行不可变借用来获取参数，再将该参数用于一个会对v进行可变借用的调用。然而，在 Rust `1.59.0` 版本中，第 2 行可以成功编译，但第 3 行无法通过借用检查器[9]。

学习和使用 Rust 所有权类型系统之所以困难，几乎肯定还有许多其他显而易见或微妙、影响重大或细微的原因，但从这里调查的文献来看，这三点（Rust 采用了不同的范式、错误消息缺乏帮助，以及借用检查器尚不完善）最为明显。

4.3 开发者为何仍选择 Rust?

> 无需手动调用 `pkg-config`，也不必借助 Autotools 宏来处理头文件和库文件的包含路径，更不必基本上依赖用户确保已安装正确版本的库；你只需编写一个`Cargo.toml` 文件，列出依赖项的名称和版本即可。 [...] 运行`cargo build`，一切就会自动完成。
>
> — [Mena-Quintero 2018][10]

我在这里要探讨的最后一个问题是：既然所有权有这么多难学、难用的原因，开发者为何仍选择使用 Rust?

或许，对此问题的第一个回答是：它们并没有。尽管自 2016 [Stack Overflow 2016以来，在每一年的 StackOverflow 调查中，2017、2018、2019，2020, 2021], Rust 都是“最受喜爱”的编程语言，但它的用户群规模仍然很小，增长也很缓慢。在同一系列调查中，Rust 于 2019 年仅以 3.2% 的占比进入最受欢迎语言榜单 [Stack Overflow 2019], , 在最近一次调查中增长到 7.03% [Stack Overflow 2021]。相比之下，Go 编程语言（常被拿来与 Rust 这门现代系统编程语言进行比较）在 2019 年的占比就已达到 8.2%，不过到 2021 年也只增长到 9.55%。类似地，TIOBE 指数将 Rust 排在 26 [TIOBE SoftwareBV 2022],，而 IEEE Spectrum 将其排在 17 [Cass 等人 2022],；相比之下，Go 的排名分别为 13 和 8。

[8]我应该提到，`iter_mut` 会使用**不安全**机制在底层实现这一点，但由于`iter_mut` 本身是 Rust 标准库提供的安全函数，新手无需接触任何不安全代码也能轻松使用它。[9][Crichton 2020] 推测，原因在于`get_mut` 是在切片上定义的（而`Vec` 类型实现了切片），而`insert` 则直接在`Vec`上实现。他们不知道这种区别为何重要，而这只会进一步印证他们的观点。[10]引文出自 [Zeng 和 Crichton 2019]。

当然，造成这种情况的原因可能有很多，远不止所有权的可用性问题；而且这些资料并非经过同行评审的来源。

但 Rust 的受欢迎程度仍在持续增长，而不出所料，多项研究中的大多数参与者提到的主要原因，是 Rust 有望保障内存安全和线程安全 [Fulton 等人，2021；Zeng 和 Crichton 2019]。遗憾的是，这两篇论文都没有深入探讨这一点，只提到安全性是最常被提及的原因。然而，这些研究中还出现了安全性以外的其他主题，它们有趣得多。

首先，安全性固然重要，但仅有安全性还不够。 [Zeng 和 Crichton 2019] 指出，采用 Rust 最常被提及的第一和第三大好处，分别是避免运行时错误和数据竞争；而第二常被提及的好处，则是 Rust 的构建工具`cargo`，它避免了其他语言的构建工具带来的诸多问题。同样，尽管 [Fulton 等人 2021]的研究参与者最常将 Rust 的安全性列为其优点，但他们提及性能和无需垃圾回收的频率也几乎一样高。

多个研究中还反复提到另一个主题：Rust 的**不安全**特性。两项调查 Rust 代码库中 **不安全**用法的研究发现，不安全代码十分常见。 [Evans 等人 2020] 调查了 `crate.io` （Rust 的在线库注册表）上所有公开可用的 Rust 库，发现 29% 的库中包含显式的**不安全**代码块。将结果筛选为最受欢迎的库（占下载量的 90%）后，这一比例升至 52.5%。在他们调查的 5 个应用中， [Qin 等人 2020] 发现了 4990 处 **不安全**用法，标准库中另有 1581 个不安全代码区域，并据此得出结论：不安全代码得到了“广泛”使用。尽管他们指出，在许多情况下这“不可避免”，而且“通常有其正当理由”，例如与用其他不安全语言（如 C）编写的现有库交互，以及将性能提升 4 到 5 倍。有趣的是，他们还发现，有时即使代码本身是安全的、无需不安全代码块也能编译，开发者仍会使用 **不安全**关键字来警示开发者。尝试采用 Rust 的人也提到了不安全代码的多种用途，指出它对于通过外部函数接口将 Rust 集成到现有代码库、访问硬件以及提升性能都是必要的 [Fulton 等人 2021]。因此，“实用主义安全性”似乎是 Rust 成功的关键因素：如果严格遵循静态保证的安全规则，许多用 Rust 编写的代码将无法实现。

同样值得注意的是，我们现在有了实证证据：尽管**不安全**被描述为一种“逃生舱”， [Evans 等人 2020], 开发者仍值得信赖，能够负责任地使用它，只在必要时使用，同时大体上维持 Rust 的安全性保证。正如 [Evans 等人 2020]发现的那样，大多数 Rust 代码库都不包含显式的不安全代码。在他们对 StackOverflow 问题的研究中， [Zhu 等人 2022] 发现，问题中的 110 个错误里，只有 3 个是通过编写不安全代码修复的。其余错误要么通过简单的安全代码修复，要么通过内部使用 unsafe 代码块、但对外提供安全接口的库函数修复（这种编程模式称为“内部不安全”）。在较大的代码库中， [Qin 等人 2020] 列举了许多与不安全代码相关的内存错误案例，但他们发现，这更多是由于类型系统（尤其是生命周期）的复杂性，而非开发者疏忽。良好的编程模式（例如内部不安全）、最佳实践（例如代码审查），以及有助于推理生命周期和不安全代码的改进工具，都能缓解造成这些错误的许多因素。

因此，尽管 Rust 不如那些缺乏其先进类型系统的同类语言受欢迎，开发者确实喜欢它，并尝试采用它，主要是因为它对安全性的承诺。除了安全性之外，他们也看重 Rust 的性能、无需垃圾回收器以及出色的工具链。尽管他们非常认可 Rust 的安全性，但仍有许多正当理由使用不安全代码，并且会负责任地这样做。

5 未来工作

在本节中，我将简要介绍人机交互和计算机科学教育领域的一些观点，我认为这些观点有助于更好地理解和改进 Rust，或者其他任何先进类型系统的可用性。这些观点较为多样，我介绍它们是为了作为产生新想法的起点，而不是提出已经充分完善的研究计划。此外，由于本报告的重点是以人为中心的方法，我不会讨论编程语言和编译器研究中的技术。不过，尤其是将这些技术与人机交互方法结合起来时，它们同样可能是改进 Rust 可用性的关键。

5.1 更好的错误消息

第4节带来的最直接、可付诸行动的启示或许是，Rust的错误消息是一项重要限制。但矛盾的是，社区普遍认为，Rust 的错误消息比大多数语言的更有帮助 [Fulton 等人 2021]。因此，提升 Rust 可用性的一个好起点，也是设计任何具有如此复杂类型的语言时需要重点考虑的问题，就是了解 Rust 的错误消息在哪些方面表现出色，又在哪些方面存在不足。

对此，一个很好的起点是 [Becker 等人 2019]。他们综述了过去 50 年来有关编译器错误消息的所有研究，并提出了许多关于这一主题的重要见解。首先，他们讨论了实证证据，表明无论是新手还是专家，程序员确实会阅读编译器错误消息 [Barik 等，2017；Prather 等人 2017]，因此，改进错误报告确实值得投入。

他们还汇编了一份基于实证研究的错误消息设计指南，这份指南可作为共同基础，用于综合各种关于 Rust 可用性的研究成果。例如，它可以为实证分析（如 [Zhu 等人 2022] 的研究）提供互补的理论背景；这些分析认为 Rust 的错误消息设计得尤为出色[11]。

这也是一份很好的资源，有助于理解 Rust 错误消息为何会失效，以及如何改进这些消息。例如，错误设计的一项重要指南是直接在消息中包含相关错误背景，而多项研究指出，Rust 有时未能做到这一点 [Blaser，2019；Dominik，2018；Zhu 等人 2022]。

根据 [Coblenz 等人 2021]关于 Rust 诊断信息缺少必要架构提示的发现，这些指南还有助于为编程语言或机器学习技术注入以人为中心的设计要素，使其能够识别此类架构变更并呈现给程序员。

5.2 程序可视化工具

人们常提到，提升 Rust 所有权可用性的下一步是将 Rust 的生命周期可视化。在讨论如何解决借用检查器不完善所带来的可用性问题时，[Crichton 2020] 建议进一步研究如何将借用检查器提供的各种静态信息可视化，并指出，要以“简洁、不打扰用户却又信息丰富”的方式呈现大量信息十分困难。

同样，[Qin 等人 2020] 提出了一个用于可视化锁生命周期的 IDE 插件。他们发现，多线程代码中一个特别常见的死锁漏洞原因是，在 Rust 中，解锁会在加锁值被丢弃时隐式发生。为此，他们建议使用“插件来突出显示 Rust 的隐式解锁位置”，这属于可视化生命周期的一个特例。

[11]深入探讨这一问题超出了本报告的范围，但根据我的个人经验，Rust 大多数时候都遵循 [Becker 等人 2019] 提出的大部分准则。

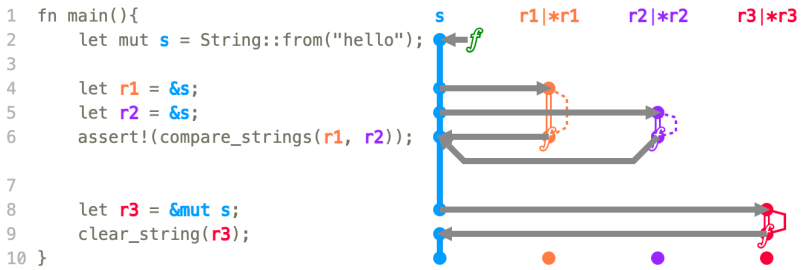

图4。RustViz 可视化示例。将鼠标悬停在右侧的节点和箭头上时，会弹出一个包含事件详细信息的窗口。

目前已有三项尝试在 Rust 中可视化所有权和生命周期的工作，但都存在局限。第一项是 RustViz [Gongming 等人 2020], 开发的一种用于构建 Rust 程序中所有权和借用事件可视化的工具。RustViz 不会自动生成这些可视化内容，而是要求“教师”通过注释待可视化的代码，并使用 Rust 库来生成可视化结果，这需要（可以说是相当大的）投入。其优点在于，生成的可视化内容具有交互性：“学习者”可以将鼠标悬停在图像的各个部分上，以获取对相关事件的简要说明。有关 RustViz 可视化效果的示例，请参见图 4。

另外两篇是相关的学士论文 [Blaser，2019；Dominik 2018]。第一篇论文中，[Dominik2018], 作者使用 Polonius（一种以 Datalog 实现借用检查规则的实验性实现）从给定的程序中提取生命周期约束，并将其可视化为有向图。他们的理由是，`rustc` 的错误消息并不总是提供有关所有权错误的全部必要信息，而这种可视化可以解决这一问题。

[Blaser 2019]随后扩展了 [Dominik 2018] 的工作，以提高其可用性。他们定义了一种算法，将完整的图筛选为一条路径，其中包含与特定错误消息相关的所有约束，如 图5所示。他们还创建了一个名为“Rust Life Assistant”的 Visual Studio Code 扩展，用于在集成开发环境中显示这些图，并设计了一种算法，将图转换为英文项目符号列表，以解释错误原因。

所有这些工作都相当有限，而且至关重要的是，它们都没有经过用户评估[12]。因此，下一步可以设计、实现并评估一种用于可视化所有权和生命周期的工具。这种工具可以用于教学（类似于 RustViz），也可以用于实用目的（类似于 Rust Life Assistant），但正如 [Zhu 等人 2022] 所指出的，“学习 Rust 是一个持续的过程”，我建议参考现有的教育和程序可视化系统相关工作，为此类工具的设计和评估提供依据。

关于这类系统的一篇优秀综述见 [Sorva 等人 2013]。除了探讨 40 多种程序可视化工具外，他们还提出了可视化系统分类法，并强调用户如何参与可视化工具的使用，与工具本身的设计同样重要。具体而言，他们引用了 [Hundhausen 等人 2002] 的研究。该研究对 24 种算法可视化（AV）工具进行了元研究，并发现：“AV 技术所采用的学习练习形式，实际上比 AV 技术生成的可视化效果质量更重要。”遗憾的是，大多数工具都聚焦于程序的运行时行为和取值，但它们仍有助于我们更深入地思考可视化系统的设计及用户与之的互动。

[12][Gongming 等人 2020] 提出了一项研究方案，但截至本报告撰写时，尚未有该研究的结果发表。

```
函数主函数() {
    let mut x = 4;
    let y = foo(&x);
    令z  = bar(&y);
    令w  = foobar(&z);
    // ...
    x = 5;
    获取(w);
}

fn foo<T>(p: T) -> T { p }
函数 bar<T>(p: T) -> T { p }
函数 foobar<T>(p: T) ->T { p }
fn获取<T>(p: T) { 未实现！() }
```

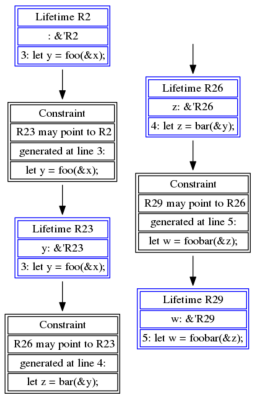

图5. Rust Life Assistant 可视化，展示代码中引用的生命周期及相应约束。

5.3 扎根理论

第4节中的研究采用了各种各样的研究方法，包括调查现有的经验报告、开展半结构式访谈、在线调查和受控实验。但迄今为止，这些研究的方法论仍存在一个明显的空白：缺乏对 Rust 程序员实际编写代码方式的深入定性理解。诸如他们使用哪些工具、是否以及如何阅读错误消息、如何推理各种错误、常见的调试策略等问题，都不在现有研究的范围内。为填补这一空白，我建议围绕 Rust 程序员的编码方式建立扎根理论，并将其与使用其他语言，或采用不同类型系统或内存管理系统的程序员进行比较。

扎根理论（GT）于20世纪60年代作为社会学的一种研究方法兴起，此后已成为许多领域中一种标准的定性研究方法，通过观察自下而上地发展理论 [Charmaz 和 Bryant 2010]。广义而言，GT作为一种方法，是一个迭代过程：通过访谈和观察收集数据，并利用开放式编码发展出一种扎根于经验数据的理论（而非受现有理论启发或用于验证现有理论）。但上述描述过于简单。GT存在多个不同版本，每个版本对知识的本质（实证主义与建构主义）都有不同的假设，其遵循的具体步骤也略有不同，且不在本报告的讨论范围内。

尽管如此，扎根理论（GT）仍是软件工程研究中一种成熟且广受欢迎的方法 [Stol 等人 2016], 最近还 [Lubin 和 Chasins 2021] 采用了建构主义扎根理论 [Charmaz 2006]，以深入了解静态类型函数式程序员如何编写代码。这正是我建议构建一种关于“Rust 程序员如何编写代码”的扎根理论的原因。推测我们可能从中学到什么，与扎根理论的理念背道而驰。不过，我希望这样的扎根理论能够帮助奠定基础，从而更好地激发我们对 Rust 及类似高级类型系统所涉及的人本研究与设计选择的兴趣，并帮助我们理解这些选择。

致谢

我要感谢我的导师 Sorin Lerner 和 Nadia Polikarpova 对本报告提出的反馈，以及他们给予我的建议和指导。我还要感谢 Hila Peleg，在我研究生阶段的前两年担任我事实上的导师；同时感谢 Ruanqianqian (Lisa) Huang 和 Shraddha Barke，感谢她们成为出色的合作伙伴和共同作者。她们的研究一直是我自身工作的重要动力和灵感来源。

引用

Jonathan Aldrich、Valentin Kostadinov 和 Craig Chambers。2002。用于程序理解的别名注解。载于第17届 ACM SIGPLAN面向对象编程、系统、语言与应用会议论文集（美国华盛顿州西雅图）（OOPSLA '02）。计算机协会，美国纽约州纽约，311–330。https: //doi.org/10.1145/582419.582448 Titus Barik、Justin Smith、Kevin Lubick、Elisabeth Holmes、Jing Feng、Emerson Murphy-Hill 和 Chris Parnin。2017。开发者会阅读编译器错误消息吗？载于第39届国际软件工程会议论文集（阿根廷布宜诺斯艾利斯）（ICSE '17）。IEEE出版社，575–585。https://doi.org/10.1109/ICSE.2017.59 Brett A. Becker、Paul Denny、Raymond Pettit、Durell Bouchard、Dennis J. Bouvier、Brian Harrington、Amir Kamil、Amey Karkare、Chris McDonald、Peter-Michael Osera、Janice L. Pearce 和 James Prather。2019。编译器错误消息被认为毫无帮助：基于文本的编程错误消息研究概览。载于计算机科学教育创新与技术工作组报告论文集（ITiCSE-WGR '19）。计算机协会，美国纽约州纽约，177–210。https://doi.org/10.1145/3344429.3372508 Jean-Philippe Bernardy、Mathieu Boespflug、Ryan R. Newton、Simon Peyton Jones 和 Arnaud Spiwack。2017。线性 Haskell：高阶多态语言中的实用线性性。《ACM编程语言期刊》2，POPL（2017年12月），5:1–5:29。https://doi.org/10.1145/3158093 David Blaser。2019。Rust 中复杂生命周期错误的简单解释。https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/David_Blaser_BA_Report.pdf Chandrasekhar Boyapati、Robert Lee 和 Martin Rinard。2002。用于安全编程的所有权类型：防止数据竞争和死锁。载于第17届 ACM SIGPLAN面向对象编程、系统、语言与应用会议论文集（美国华盛顿州西雅图）（OOPSLA '02）。计算机协会，美国纽约州纽约，211–230。https://doi.org/10.1145/582419.582440 Chandrasekhar Boyapati、Barbara Liskov 和 Liuba Shrira。2003a。用于对象封装的所有权类型。载于第30届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（美国路易斯安那州新奥尔良）（POPL '03）。计算机协会，美国纽约州纽约，213–223。https://doi.org/10.1145/604131.604156 Chandrasekhar Boyapati、Alexandru Salcianu、William Beebee 和 Martin Rinard。2003b。用于实时 Java 中安全区域内存管理的所有权类型。载于 ACM SIGPLAN 2003编程语言设计与实现会议论文集（美国加利福尼亚州圣迭戈）（PLDI '03）。计算机协会，美国纽约州纽约，324–337。https://doi.org/10.1145/781131.781168 John Boyland。2001。别名隐藏：没有破坏性读取的唯一变量。《软件：实践与经验》31，6（2001），533–553。https://doi.org/10.1002/spe.370 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.370 Stephen Cass、Preeti Kulkarni 和 Erico Guizzo。2022。2021年顶级编程语言。https://spectrum.ieee.org/top-programming-languages/ Kathy Charmaz。2006。构建扎根理论：定性分析实用指南。SAGE出版集团有限公司 Kathy。Charmaz 和 Antony Bryant。2010。扎根理论。载于《国际教育百科全书》（第三版），Penelope Peterson、Eva Baker 和 Barry McGaw（编）。爱思唯尔，牛津，406–412。https://doi.org/10.1016/B978-0-08- 044894-7.01581-5 Dave Clarke 和 Tobias Wrigstad。2003。外部唯一性已经足够唯一。载于 ECOOP 2003——面向对象编程，Luca Cardelli（编）。Springer Berlin Heidelberg，柏林、海德堡，176–200。Dave Clarke、Johan Östlund、Ilya Sergey 和 Tobias Wrigstad。2013。所有权类型：综述。载于《面向对象编程中的别名：类型、分析与验证》，Dave Clarke、James Noble 和 Tobias Wrigstad（编）。施普林格，柏林、海德堡，15–58。https://doi.org/10.1007/978-3-642-36946-9_3 David G. Clarke、John M. Potter 和 James Noble。1998。用于灵活别名保护的所有权类型。载于第13届 ACM SIGPLAN面向对象编程、系统、语言与应用会议论文集（加拿大不列颠哥伦比亚省温哥华）（OOPSLA '98）。计算机协会，美国纽约州纽约，48–64。https: //doi.org/10.1145/286936.286947 Michael Coblenz、Jonathan Aldrich、Brad A. Myers 和 Joshua Sunshine。2018。跨学科编程语言设计。载于2018年 ACM SIGPLAN关于编程与软件的新思想、新范式与反思国际研讨会论文集（美国马萨诸塞州波士顿）（Onward! 2018）。计算机协会，133–146。https://doi.org/10.1145/3276954.3276965 Michael Coblenz、Michelle Mazurek 和 Michael Hicks。2021。Bronze 垃圾回收器是否让 Rust 更易于使用？一项受控实验。arXiv预印本 arXiv:2110.01098（2021）。Will Crichton。2018。系统编程究竟是什么？https://willcrichton.net/notes/systems-programming/ Will Crichton。2020。所有权的可用性。arXiv预印本 arXiv:2011.06171（2020）。Will Crichton、Marco Patrignani、Maneesh Agrawala 和 Pat Hanrahan。2021。通过所有权实现模块化信息流。CoRR abs/2111.13662（2021）。arXiv:2111.13662 https://arxiv.org/abs/2111.13662

Matthew Davis、Peter Schachte、Zoltan Somogyi 和 Harald Søndergaard。2012年。面向 Go 的基于区域的内存管理。在 2012年ACM SIGPLAN研讨会“内存系统性能与正确性”（中国北京）（MSPC ’12）论文集。计算机协会，美国纽约州纽约，美国，58–67。https: //doi.org/10.1145/2247684.2247695 Dan Diaper。2004年。理解人机交互的任务分析。《人机交互任务分析手册》（2004），5–47。Dietler Dominik。2018年。Rust生命周期约束可视化。https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dominik_Dietler_BA_report.pdf Ana Nora Evans、Bradford Campbell 和 Mary Lou Soffa。2020年。软件开发人员是否安全地使用 Rust？在第42届ACM/IEEE国际软件工程大会（ICSE ’20）论文集。计算机协会，美国纽约州纽约，美国，246–257。https://doi.org/10.1145/3377811.3380413 Manuel Fahndrich 和 Robert DeLine。2002年。采用与聚焦：命令式编程的实用线性类型。在 2002年ACM SIGPLAN编程语言设计与实现大会（德国柏林）（PLDI ’02）论文集。计算机协会，美国纽约州纽约，美国，13–24。https://doi.org/10.1145/512529.512532 Kelsey R Fulton、Anna Chan、Daniel Votipka、Michael Hicks 和 Michelle L Mazurek。2021年。采用安全编程语言的益处与缺点：以 Rust 为案例研究。在第十七届可用隐私与安全研讨会（SOUPS 2021）。597–616。Gongming、Luo、Vishnu Reddy、Marcelo Almeida、Yingying Zhu、Ke Du 和 Cyrus Omar。2020年。RustViz：交互式可视化所有权与借用。arXiv:2011.*09012 [*cs*]*（2020年11月）。http://arxiv.org/abs/2011.09012 arXiv：2011.09012。Dan Grossman、Michael Hicks、Trevor Jim 和 Greg Morrisett。2005年。Cyclone：C语言的类型安全方言。C/C++ Users Journal 23，1（2005），112–139。Dan Grossman、Greg Morrisett、Trevor Jim、Michael Hicks、Yanling Wang 和 James Cheney。2002a年。Cyclone中的基于区域的内存管理。在 2002年ACM SIGPLAN编程语言设计与实现大会（德国柏林）（PLDI ’02）论文集。计算机协会，美国纽约州纽约，美国，282–293。https://doi.org/10.1145/512529.512563 Dan Grossman、Greg Morrisett、Trevor Jim、Michael Hicks、Yanling Wang 和 James Cheney。2002b年。Cyclone中的基于区域的内存管理。ACM SIGPLAN通讯 37，5（2002年5月），282–293。https://doi.org/10.1145/543552.512563 Philipp Haller 和 Martin Odersky。2010年。唯一性与借用能力。在 ECOOP 2010——面向对象编程，Theo D’Hondt（编）。Springer Berlin Heidelberg，柏林，海德堡，354–378。Eric Holk、Ryan Newton、Jeremy Siek 和 Andrew Lumsdaine。2014年。GPU编程语言的基于区域的内存管理：在精简主机上实现丰富数据结构。在 2014年ACM面向对象编程系统、语言与应用国际会议（美国俄勒冈州波特兰）（OOPSLA ’14）论文集。计算机协会，美国纽约州纽约，美国，141–155。https://doi.org/10.1145/2660193.2660244 Christopher D. Hundhausen、Sarah A. Douglas 和 John T. Stasko。2002年。算法可视化有效性的元研究。《可视语言与计算期刊》13，3（2002），259–290。https://doi.org/10.1006/jvlc.2002.0237 Herman Ruge Jervell。1996年。Jean-Yves Girard。线性逻辑。《理论计算机科学》，第50卷（1987），第1–101页。— A. S. Troelstra。《线性逻辑讲义》。CSLI讲义，第29号。语言与信息研究中心，斯坦福，1992年；亦由剑桥大学出版社发行，纽约，ix，200页。《符号逻辑期刊》61，1（1996），336–338。https://doi.org/10.2307/2275616 Trevor Jim、J. Morrisett、Dan Grossman、Michael Hicks、James Cheney 和 Yanling Wang。2002年。Cyclone：C语言的安全方言。《2002年USENIX年度技术会议论文集》，275–288。Steve Klabnik 和 Carol Nichols。2017年。《Rust程序设计语言》。No Starch Press，美国加利福尼亚州旧金山。Justin Lubin 和 Sarah E. Chasins。2021年。静态类型函数式程序员如何编写代码。ACM编程语言期刊 5，OOPSLA（2021年10月），155:1–155:30。https://doi.org/10.1145/3485532 Henning Makholm。2000年。Prolog中的基于区域的内存管理。技术报告。在第2届国际内存管理研讨会论文集。Nicholas D. Matsakis 和 Felix S. Klock。2014年。Rust语言。在 2014年ACM SIGAda高完整性语言技术年度会议（美国俄勒冈州波特兰）（HILT ’14）论文集。计算机协会，美国纽约州纽约，美国，103–104。https://doi.org/10.1145/2663171.2663188 Joseph Mcgrath，E.。1995年。方法论至关重要：行为科学与社会科学研究。在《人机交互读本》中，Ronald Baecker、M.、Jonathan Grudin、William Buxton、A.S. 和 Saul Greenberg（编）。Morgan Kaufmann，152–169。https://doi.org/10.1016/B978-0-08-051574-8.50019-4 Federico Mena-Quintero。2018年。我在 C 中怀念的 Rust 特性。https://web.archive.org/web/20210119090858/https://people.gnome. org/~federico/blog/rust-things-i-miss-in-c.html Greg Morrisett、Karl Crary、Neal Glew、Dan Grossman、Richard Samuels、Frederick Smith、David Walker、Stephanie Weirich 和 Steve Zdancewic。1999年。TALx86：一种现实的类型化汇编语言。在第二届 ACM SIGPLAN 研讨会“系统软件的编译器支持”。

系统软件的编译器支持。美国佐治亚州亚特兰大，25–35。Khanh Nguyen、Lu Fang、Guoqing Xu 和 Brian Demsky。2015年。面向大数据系统的推测式区域内存管理。在第8届编程语言与操作系统研讨会（加利福尼亚州蒙特雷）（PLOS '15）论文集。计算机协会，美国纽约州纽约，美国，27–32。https://doi.org/10.1145/2818302.2818308 Benjamin C. Pierce。2004年。《类型与编程语言高级专题》。麻省理工学院出版社。James Prather、Raymond Pettit、Kayla Holcomb McMurry、Alani Peters、John Homer、Nevan Simone 和 Maxine Cohen。2017年。新手与编译器错误消息的交互：一种人因方法。在 2017年ACM国际计算机教育研究会议（华盛顿州塔科马，美国）（ICER '17）论文集。计算机协会，美国纽约州纽约，美国，74–82。https://doi.org/10.1145/3105726.3106169 Boqin Qin、Yilun Chen、Zeming Yu、Linhai Song 和 Yiying Zhang。2020年。真实世界 Rust 程序中的内存与线程安全实践及问题。在第41届ACM SIGPLAN编程语言设计与实现会议（英国伦敦）（PLDI 2020）论文集。计算机协会，美国纽约州纽约，美国，763–779。https://doi.org/10.1145/3385412.3386036 Rust 项目开发者。2022。《Rustonomicon》。https://doc.rust-lang.org/nomicon/ Nischal Shrestha、Colton Botta、Titus Barik 和 Chris Parnin。2020年。我们又来了：为什么开发者难以学习另一种编程语言？在第42届ACM/IEEE国际软件工程大会（韩国首尔）（ICSE '20）论文集。计算机协会，美国纽约州纽约，美国，691–701。https://doi.org/10.1145/3377811.3380352 Juha Sorva、Ville Karavirta 和 Lauri Malmi。2013年。入门编程教育通用程序可视化系统综述。ACM计算教育学汇刊 13，4（2013年11月），1–64。https://doi.org/10.1145/2490822 Stack Overflow。2016年。Stack Overflow 2016年开发者调查结果。https://insights.stackoverflow.com/survey/2016 Stack Overflow。2017年。Stack Overflow 2017年开发者调查。https://insights.stackoverflow.com/survey/2017/?utm_ source=so-owned&utm_medium=social&utm_campaign=dev-survey-2017&utm_content=social-share Stack Overflow。2018年。Stack Overflow 2018年开发者调查。https://insights.stackoverflow.com/survey/2018/?utm_ source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share Stack Overflow。2019年。Stack Overflow 2019年开发者调查。https://insights.stackoverflow.com/survey/2019/?utm_ source=social-share&utm_medium=social&utm_campaign=dev-survey-2019 Stack Overflow。2020年。Stack Overflow 2020年开发者调查。https://insights.stackoverflow.com/survey/2020/?utm_ source=so-owned&utm_medium=social&utm_campaign=dev-survey-2020 Stack Overflow。2021年。Stack Overflow 2021年开发者调查。https://insights.stackoverflow.com/survey/2021/?utm_ source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 Klaas-Jan Stol、Paul Ralph 和 Brian Fitzgerald。2016年。软件工程研究中的扎根理论：批判性综述与指南。在第38届国际软件工程会议（ICSE '16）论文集。计算机协会，美国纽约州纽约，美国，120–131。https://doi.org/10.1145/2884781.2884833 Rust 核心团队。2018年。宣布 Rust 1.31 和 Rust 2018。https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust- 2018.html Rust 调查团队。2020年。Rust 2020年调查结果。https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html TIOBE Software BV。2022年。索引。https://www.tiobe.com/tiobe-index/ Mads Tofte 和 Jean-Pierre Talpin。1997年。基于区域的内存管理。信息与计算 132，2（1997），109–176。Philip Wadler。1990年。线性类型可以改变世界！在《编程概念与方法》中。David Walker 和 Kevin Watkins。2001年。区域与线性类型（扩展摘要）。在第六届 ACM SIGPLAN国际函数式编程会议（意大利佛罗伦萨）（ICFP '01）论文集。计算机协会，美国纽约州纽约，美国，181–192。https://doi.org/10.1145/507635.507658 Aaron Weiss、Olek Gierczak、Daniel Patterson 和 Amal Ahmed。2021年。Oxide：Rust 的本质。arXiv:1903.*00982 [*cs*]*（2021年10月）。http://arxiv.org/abs/1903.00982 arXiv：1903.00982。Hongwei Xi 和 Frank Pfenning。1999年。实用编程中的依赖类型。在第26届ACM SIGPLAN-SIGACT编程语言原理研讨会（美国德克萨斯州圣安东尼奥）（POPL '99）论文集。计算机协会，美国纽约州纽约，美国，214–227。https://doi.org/10.1145/292540.292560 Anna Zeng 和 Will Crichton。2019年。通过在线讨论识别 Rust 采用的障碍。arXiv预印本 arXiv:1901.01001（2019）。Shuofei Zhu、Ziyi Zhang、Boqin Qin、Aiping Xiong 和 Linhai Song。2022年。Rust 的学习与编程挑战：一项混合方法研究。（2022），13。https://doi.org/10.1145/3510003.3510164